DOCUMENT RESUME

ED 288 505 IR 012 901

AUTHOR Eisenstadt, Marc; Brayshaw, Mike
TITLE An Integrated Textbook, Video, and Software Environment for Novice and Expert Prolog Programmers. Technical Report No. 23.
INSTITUTION Open Univ., Milton Keynes, (England). Human Cognition Research Lab.
PUB DATE Jun 87
NOTE 22p.; Paper presented at the International Conference of the Prolog Education Group (2nd, Exeter, England, July 8-10, 1987).
PUB TYPE Viewpoints (120) -- Reports - Research/Technical (143)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS Animation; Computer Software; *Expert Systems; Foreign Countries; Literature Reviews; *Man Machine Systems; *Models; *Programing; *Teaching Methods; Videotape Recordings
IDENTIFIERS *PROLOG Programing Language

ABSTRACT

This paper describes a Prolog execution model which serves as the uniform basis of textbook material, video-based teaching material, and an advanced graphical user interface for Prolog programmers. The model, based upon an augmented AND/OR tree representation of Prolog programs, uses an enriched "status box" in place of the traditional AND/OR tree node to show precise details of clause head/body processing and to deal correctly with the "cut." The purpose of this integrated environment is to address two hitherto unreconciled goals: (1) to provide a clear and consistent model of Prolog program execution for novices that can be used for teaching, exercises, and self-paced study; and (2) to provide an advanced graphical interface for logic programming that is usable by Prolog experts in real world application domains, and that provides no conceptual change from (1). The end result comprises a textbook/workbook combination which enables students to work through "paper-and-pencil" execution snapshots; video animation sequences showing detailed run-throughs of moderately complex Prolog programs; and a working graphical tracer/debugger currently running on Apollo workstations. Seventeen references are listed. (Author/RP)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED288505

The Open University

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☒ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

An Integrated Textbook, Video, and Software Environment for Novice and Expert Prolog Programmers

Marc Eisenstadt
Mike Brayshaw

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

E.S. Henderson

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

Human Cognition Research Laboratory
Milton Keynes MK7 6AA, U.K.

Technical Report No. 23
June, 1987

IR012901

An integrated textbook, video, and software environment for novice and expert Prolog programmers[1]

Marc Eisenstadt and Mike Brayshaw

HCRL

Human Cognition Research Laboratory
The Open University
Milton Keynes MK7 6AA, UK

Abstract: The paper describes a Prolog execution model which serves as the uniform basis for textbook material, video-based teaching material, and an advanced graphical user interface for Prolog programmers. The model, based upon an augmented AND/OR tree representation of Prolog programs, uses an enriched 'status box' in place of the traditional AND/OR tree node to show precise details of clause head/body processing and to deal correctly with the 'cut'. The purpose of our integrated environment is to address two hitherto unreconciled goals: (i) to provide a clear and consistent model of Prolog program execution for novices that can be used for teaching, exercises, and self-paced study, and (ii) to provide an advanced graphical interface for logic programming that is usable by Prolog experts in real world application domains and that requires no conceptual change from (i). The end result comprises (a) a textbook/workbook combination which enables students to work through 'paper-and-pencil' execution snapshots; (b) video animation sequences showing detailed run-throughs of moderately complex Prolog programs, and (c) a working graphical tracer/debugger currently running on Apollo workstations.

Topic Areas: Teaching Logic Programming, Human-Machine Interface, Curriculum Issues.

1 Introduction

1.1 Motivation

Existing Prolog textbooks (e.g. Clocksin & Mellish, 1984; Bratko, 1986; Sterling & Shapiro, 1986) provide adequate descriptions of the language, yet novices still 'come unstuck' when having to write or debug Prolog programs that exhibit anything other than the most trivial forms of backtracking and unification (Coombs & Stell, 1985; Taylor, 1987). It could be that the language is just inherently 'tricky', and requires lots of practice. Alternatively, it might be the case that novices are never provided with (or at least never acquire) a clear execution model, and this leads them astray. The analysis of Bundy et. al. (1986) suggested that texts such as Clocksin & Mellish (1st edition, 1981) contained several different and potentially conflicting 'stories' of Prolog execution, and that this was a source of possible confusion. To their credit, Clocksin & Mellish provided a much more coherent story of Prolog execution in the second edition of their book (1984). Nevertheless, there is a rather wide gulf between what is presented in any existing text and what is observed when a real Prolog program 'acts funny' (Taylor, 1987).

The importance of clear models of the underlying machine has been highlighted elsewhere (e.g. Du Boulay, O'Shea & Monk, 1981;Halasz and Moran,1982). Here we present a notional machine that is based directly on a meta-level interpreter, thereby allowing us to follow execution steps in very precise detail. The advantages that this conceptual model gives us are twofold:

[1]To appear in *Proceedings of the 2nd International Conference of the Prolog Education Group (PEG-87)*, Exeter, UK, 8th-10th July, 1987. This work was supported by UK SERC/Alvey Research grants GR/C/69344 & GR/E/2333, plus a grant from the SERC Information Technology Training Initiative. The graphical tracer is being developed as part of Alvey IKBS Project number 161, conducted jointly with Expert Systems International. We would particularly like to acknowledge the following for their valuable assistance: Malcolm Dixon, Hank Kahney, Frank Kriwaczek, and Peter Ross for co-authoring/revising the course notes; David Thomas, Chris Hogger, and numerous other members of the UK Logic Programming community for their original efforts in helping to launch the Prolog course; Glan Thomas and Mike Peat for producing the videos; Malcolm Story and Adam Gawronski for implementing the OU Prolog interpreter.

(i) Far from inundating the novice with 'too much truth', we feel on the contrary that describing the machine at such a level makes the basic concepts of the language unambiguous, more readily accessible, easier to understand, and hence less error-prone.
(ii) We can present a story directly based upon the machine rather than introducing extra artificial layers of description and metaphor (e.g. Carroll and Thomas,1982).

Our focus has been on producing an account of the language which is rich enough to satisfy the needs of experts but has the descriptive power and clarity for teaching difficult concepts to a novice programmmer. Our work has developed from two ends simultaneously. That is, we were developing video-based teaching material for novice Prolog programmers at the same time as we were implementing graphics facilities for helping experts observe a 2- or 3-thousand node search space. Only by forcing these two paths to converge could we cater for the 'upwardly mobile student' who learned about Prolog in the early phases and then went on to become a serious Prolog user.

1.2 Pedagogically-motivated tracers

Amongst the numerous tracing and debugging packages built for Prolog and other languages, several have been motivated explicitly by pedagogical considerations. Here we mention only the four most relevant. For a detailed overview of other work the reader is referred to the excellent summary found in Rajan (1986).

Mellish has developed an animated AND/OR tree trace which displays the run-time behaviour of small Prolog programs on VT100-style terminals connected to a machine running the Sussex University/System Designers Ltd. POPLOG environment (Hardy, 1984). This tracer was developed as part of an effort to provide a consistent 'execution story' for the student learning Prolog, and indeed the second edition of Clocksin & Mellish (1984) conforms with the style and content of the animated tracer. Two important problems with Mellish's facility are (a) variables are destructively replaced by their instantiations on the screen in 'real' (or slightly slowed down) time, so it can be hard to 'go back' and see what actually happened; (b) only toy programs can be viewed. The principle of showing an animated AND/OR tree is nevertheless fundamentally important.

Rajan's (1986) single-stepper works its way through Prolog code by highlighting relevant portions in the database at the appropriate moments, and by instantiating clause bodies in their entirety 'in place' in the code being traced. By showing unification in painstaking detail, a very clear account of execution can be presented to novices. However, Rajan's single-stepper doesn't fully show variable renaming, and can only deal with relatively small execution spaces. Although both the renaming and search space problems were not relevant for the class of beginners Rajan was addressing, they are problems which need to be resolved for longer-term users.

Brown and Sedgewick(1984) in their BALSA environment have sought to use animated displays as a way of demonstrating algorithms and teaching basic computing concepts. Although they include the important notion of a 'replay' facility in their program, the system is an algorithm viewer, rather than a true trace facility (i.e. ordinary users cannot type in original code and see a graphical trace of it).

DEWLAP is a Prolog debugger by Dewar and Cleary (1986) that graphically shows the state of the program execution by using a fisheye zoom to show detailed information about the item of current focus and gradually less about those that are further away and of less interest. The system shows individual clause head unification and allows the user to specify special ways to view individual data objects. The trace display rapidly gets very complicated even on small examples and the nature of the zooming they use means that is not possible easily to compare parts of the tree that are a large distance apart. Although Dewar and Cleary acknowledge the desirability of seeing the program's behaviour in terms of an AND/OR

tree, they compromise in the DEWLAP diagram with a resulting loss of clarity.

1.3 A way forward

We have been most influenced by the work of Rajan (1986) and Bundy et. al. (1986). We share with Rajan the desire to 'tell the truth' about Prolog execution in gory detail, and to show unification details in terms which map directly onto the user's source code. We differ from Rajan in preferring to show large execution spaces graphically rather than textually. The work of Bundy et. al. has cogently highlighted the virtues of sticking firmly with an AND/OR tree execution model in preference to other possible models, including OR trees and 'Byrd Box' models (Byrd, 1980). Our work differs from that of Bundy et. al. in three important respects: (i) in our 'close up' view we display much more detail about clause head matching and unification, but without sacrificing the basic clarity of an AND/OR tree representation; (ii) our 'long-distance' view is consistent with our 'close up' story, and is not only suitable for machine implementation, but is also capable of displaying the execution of enormous Prolog programs; (iii) we don't require separate 'resolution tables' or 'database' displays, although we naturally assume that in a modern Prolog environment the user can trivially display a window showing relevant source code.

Our work started with the following two premises: (i) it is essential both for teaching and debugging purposes to show the full execution space of large Prolog programs; (ii) it is essential to base both teaching material and a working trace package on an execution model which discriminates between clause head matching and clause body execution.

We have stuck with these two premises partly because of the challenge of resolving their underlying contradiction: premise (i) involves a global view of things, while premise (ii) involves a very close up view. Our aim has been to reconcile these differences without losing the advantages provided by either. We feel strongly that different levels of detail (i.e. different grain-size of analysis) involve different conceptual views of what is happening, and therefore a simple 'aerial view/close-up zoom' facility does *not* provide a magical solution. It is nevertheless possible to accommodate both premises, despite the commonly-held misconception that both are 'too hard' are 'too messy' to achieve. The key insights which enable this accommodation, and which drive the whole of the work described in this paper, are the following:

- It is possible to display an execution space involving thousands of nodes on today's graphics workstations.
- When a Prolog programmer is debugging a program which he or she has personally been developing over a period of weeks or months, an overall graphical view of the execution space of that program is highly meaningful to that programmer, because it conveys its own gestalt.
- The concept of a 'node' in a traditional AND/OR tree is needlessly impoverished. With just a few enhancements, and for a very small computational overhead, a simple node can become a 'status box' which concisely encapsulates a goal's history, including detailed clause head matching information.
- The traditional AND/OR tree does not reveal the difference between one clause containing a disjunction and two separate clauses. More specifically, the distinction between clause head and clause body is not shown in AND/OR trees, despite the overwhelming importance of the head/body distinction to the debugging of Prolog programs. A minor notational variant enables us to overcome this problem.

The layout of the paper is as follows: section 2 describes the basic representation and discusses various examples; section 3 outlines the teaching material; section 4 overviews the graphical debugger; section 5

contains conclusions and a discussion of the future directions of the work.

2 A model of Prolog execution

The sub-sections which follow introduce an idealised model of the Prolog machine which we dub the Transparent Prolog Machine, introduced originally in Eisenstadt & Brayshaw (1986).

2.1 The Procedure Status Box

The cornerstone of our convention, illustrated in figure 1, is the *procedure status box*. This box replaces the simple and/or tree 'node' in our display of the execution space. The upper part indicates the *goal status*, and informs us whether the goal is currently being processed (a '?'), whether it has succeeded (a tick), whether it has failed (a cross), or whether it initially succeeded and then failed upon backtracking (tick/cross combination). The lower half is a *clause counter*. The clause counter tells us which of several clauses for a given relation in the database is currently being processed. Circular nodes are used instead of procedure status boxes to depict system primitives, and may contain any of the following four symbols: question mark, tick, cross, tick/cross combination (corresponding to their meanings as above).

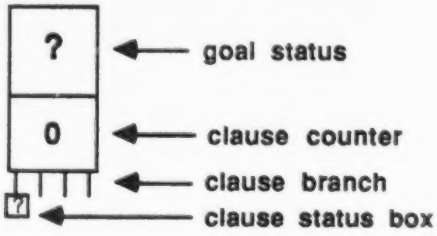

Figure 1. The procedure status box

The small branches dangling down underneath the box in figure 1 correspond exactly to the individual clauses in the database, and are therefore called *clause branches*. The leftmost clause branch corresponds to the first clause, and the next one corresponds to the second clause, etc. When there are more than five clause branches, a special 'collapsing' convention is adopted. The square boxes at the end of clause branches are *clause status boxes*. Such boxes use the same question-mark, tick, cross, and tick/cross combination to depict the status of individual clauses. If a given clause head does not unify, then a short horizontal *dead-end bar* is added instead of a clause status box. Clause branches correspond to 'or' choices, but are drawn differently from their traditional counterparts in order to make the processing of individual clauses obvious at a glance.

Consider the following four-clause database:

```
drinks(tom, wine).
drinks(john, beer).
drinks(sue, wine).
drinks(X, water). % everyone (everything) drinks water
```

Figures 2a and 2b show the states of affairs at the beginning and end of processing the query
?- drinks(sue, wine).

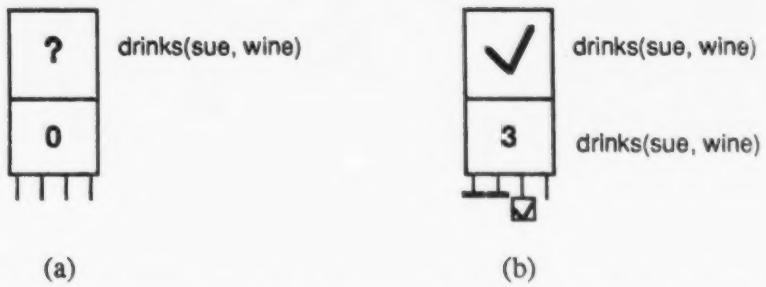

Figure 2. Processing the query **?- drinks(sue, wine)**. (a) shows the status box at the beginning of the query, and (b) shows the outcome.

2.2 Unification and renaming

To illustrate unification, the relations and arguments next to the top half of each procedure status box depict the state of play when the goal was invoked, whereas the relations and arguments next to the bottom half of each procedure status box depict the matching clause head found in the data base. User-chosen variable names are subscripted automatically to indicate renamed variables. The subscript counter is incremented for each new nonground clause invoked as a (sub)goal The diagrams use a sideways '=' with arrowheads to show unification. Up arrows indicate output variables; down arrows indicate input variables. Given the four clauses for **drinks/2** used earlier, Figure 3 shows the status box representation of the following three interactions:

a) **?- drinks(john, X).**
X = beer

b) **?- drinks(foo, water).**
yes

c) **?- drinks(foo, X).**
X = water

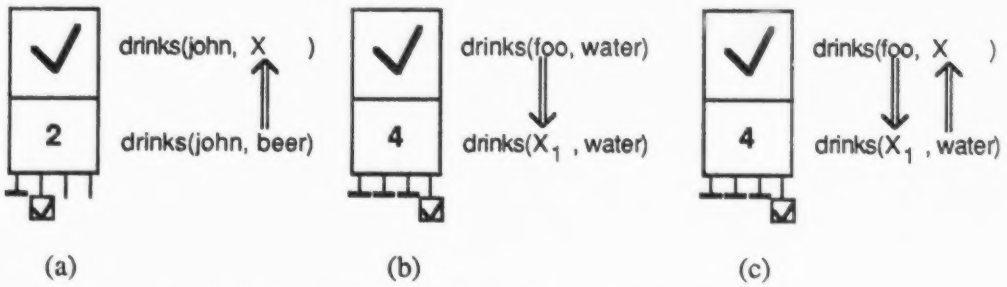

Figure 3. Unification details, showing input and output variables for three different goals.

Right-angled arrows indicate a variable 'passed across' or shared with a sister goal. Headless arrows can be used for teaching purposes to indicate directly-matching terms, particularly in cases where the match is subtle. Often there is a direct visual correspondence between a variable and the arrow showing its instantiation in the diagram. Whenever the correspondence is 'indirect', i.e. the instantiation has come 'from elsewhere', we place a small lozenge beneath the variable to show its instantiation at the moment of the execution 'snapshot'. An example of this is provided in section 2.3.

2.3 AORTA Diagrams

The full space of goals traversed during execution is described in terms of an AND/OR Tree, Augmented (hence AORTA) by the above status boxes and unification details. Consider the following program:

```
g :- d1(X), d2(X).
g :- d3(X), d4(X).

d1(a).      d2(d).  d3(f).      d4(i).
d1(b).      d2(e).  d3(g).      d4(j).
d1(c).              d3(h).      d4(h).
```

Figure 4 shows the AORTA diagram depicting the *full search space* for the query **?- g.** The clause head labels, **C1** and **C2**, are here only displayed for convenience. The display developed for teaching purposes would only show that part of the search space which is actually traversed during execution. In our graphical tracer implementation, we distinguish between the execution space which has been traversed thus far, and the execution space which we can guarantee (via behind-the-scenes analysis) will *eventually* be traversed. We call this latter space of nodes the 'pre-ordained execution space'.

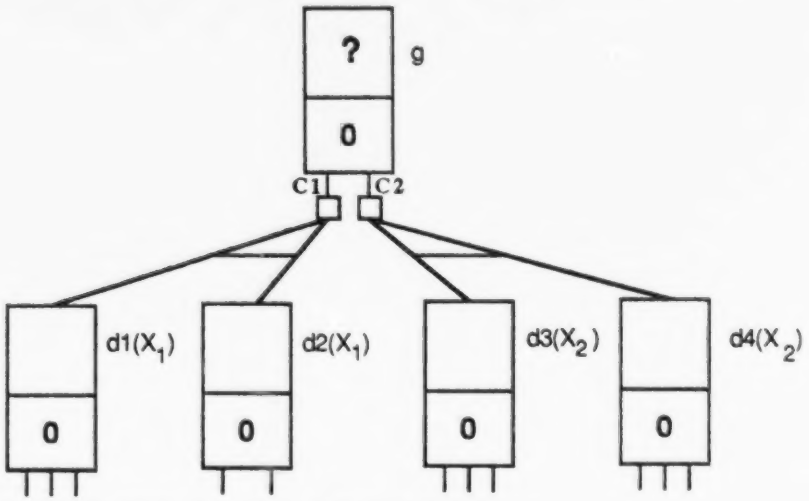

Figure 4. A sample AORTA Diagram, showing the full search space for **g**.

In order to describe clearly the actions of the Prolog interpreter, we employ a family metaphor to describe the relationships among the goals in the tree. Thus we speak of primary goals as 'mother' goals, and their subgoals as 'daughter' goals. In figure 4, for example, **d1** and **d2** are sister goals with **g** as their mother. To indicate that two subgoals are fully-fledged sisters notice that they are linked together by a 'conjunction bar'. **d3** and **d4** are also sisters with **g** as their mother, but have a different lineage from **d1** and **d2** because they are part of a different clause. Metaphorically we can account for this by attributing different fathers to **d1-d2** as compared with **d3-d4.** These fathers are illustrated by the clause branches **C1** and **C2** in the diagram. Thus, **C2** is a *step-father* of **d2**, and **d2** and **d3** are *step-sisters*. To reflect the chronology of execution, we also note that **d1** is an *older sister* of **d2** (because goal **d1** was 'born' first). Analogously, we refer to **C2** as a *future stepfather* of **d2**.

The family metaphor makes descriptions of backtracking and the cut straightforward. When the interpreter backtracks, the 'backtracking point' (i.e. the place to resume processing) is easily identified in an AORTA diagram. To do this, it is only necessary to first identify the immediately older sister of the just-failed goal. If that older sister is a primitive (a circular node), it itself is the backtracking point. Otherwise, the

backtracking point is that older sister's 'youngest surviving heir', i.e. the rightmost, most deeply-nested *clause* status box with a tick in it. When a cut is encountered, three things happen:

1) older sister goals and their descendants are frozen
2) future step-fathers are eliminated
3) the goal succeeds

'Freezing' is described as the process of enshrouding AORTA diagram boxes in a 'frozen cloud' which makes them inaccessible to backtracking during the current invocation. To illustrate this notation in action, consider the following program, which is contrived to illustrated a large number of our conventions in a small space:

```
party(X):- happy(X), birthday(X).
party(X):- friends(X,Y), sad(Y).  % have a party to cheer up a  sad friend

happy(X):-hot, humid, not raining, !, swimming(X).
happy(X):-cloudy,  watching_tv(X).
happy(X):-cloudy,  having_fun(X).

cloudy.
hot.
humid.  %today's weather

having_fun(tom).
having_fun(sam).

swimming(john).

watching_tv(john).

sad(bill).
sad(sam).

birthday(tom).
birthday(sam).

friends(tom,john).
friends(tom,sam).
```

Figures 5 and 6 represent respectively an intermediate and final AORTA 'snapshot' of the query **?- party(Name)**.

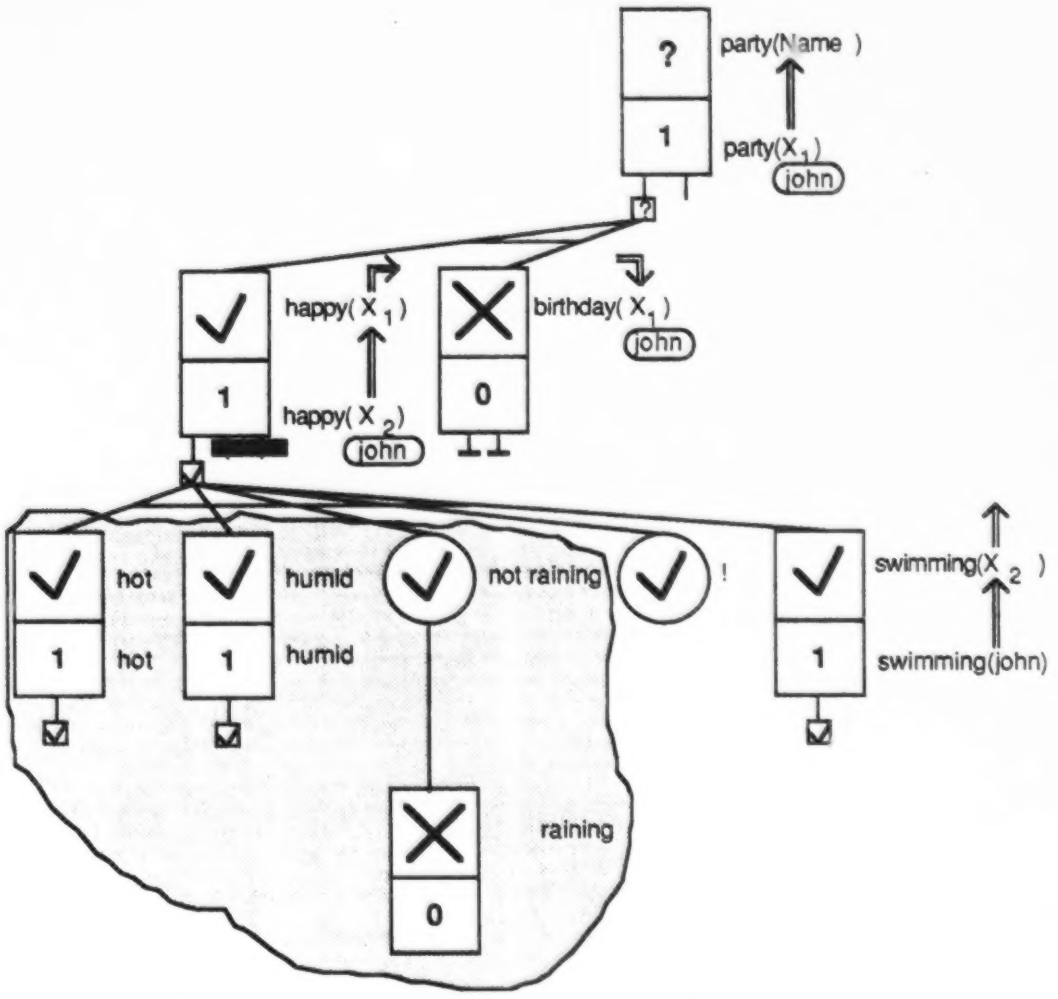

Figure 5 Intermediate AORTA snapshot after the query ?- **party(Name)**. **birthday/1** has just failed.

Let's consider the details which lead up to figure 5. For the top goal **party**, clause one is attempted, which leads to the first clause of **happy** being tried. This in turn leads to **hot** and **humid** succeeding. Notice that the clause counter indicates that both succeed on clause one, and moreover they must both be facts, otherwise we would have seen their subgoals, and they can have had only one clause each because their procedure status boxes show only a single clause branch underneath. Next, **not raining** is attempted, and succeeds because **raining** fails. There is no definition of **raining** in the database, hence no clause counter or clause branches (which indeed would cause a run-time error in many implementations of Prolog). Next, **not** and 'cut' succeed. The result of executing the cut is to freeze any of the cut's older sisters and their descendants (e.g. any subgoals like **raining**), indicated by the *frozen cloud* about them, and to eliminate any other possible future step-fathers, hence the 'strike-out' bar over the second and third clause branches of **happy**. Next, **swimming** succeeds, causing **happy** to succeed and **birthday(john)** to be attempted. **birthday(john)** fails to unify on either of its two clauses, producing the 'dead-end-bars' beneath the clause branches and a 'cross' in the top part of its procedure status box. This is the precise moment at which the snapshot in figure 5 is taken. Now let's look at the remaining steps leading up to the final execution snapshot depicted in figure 6.

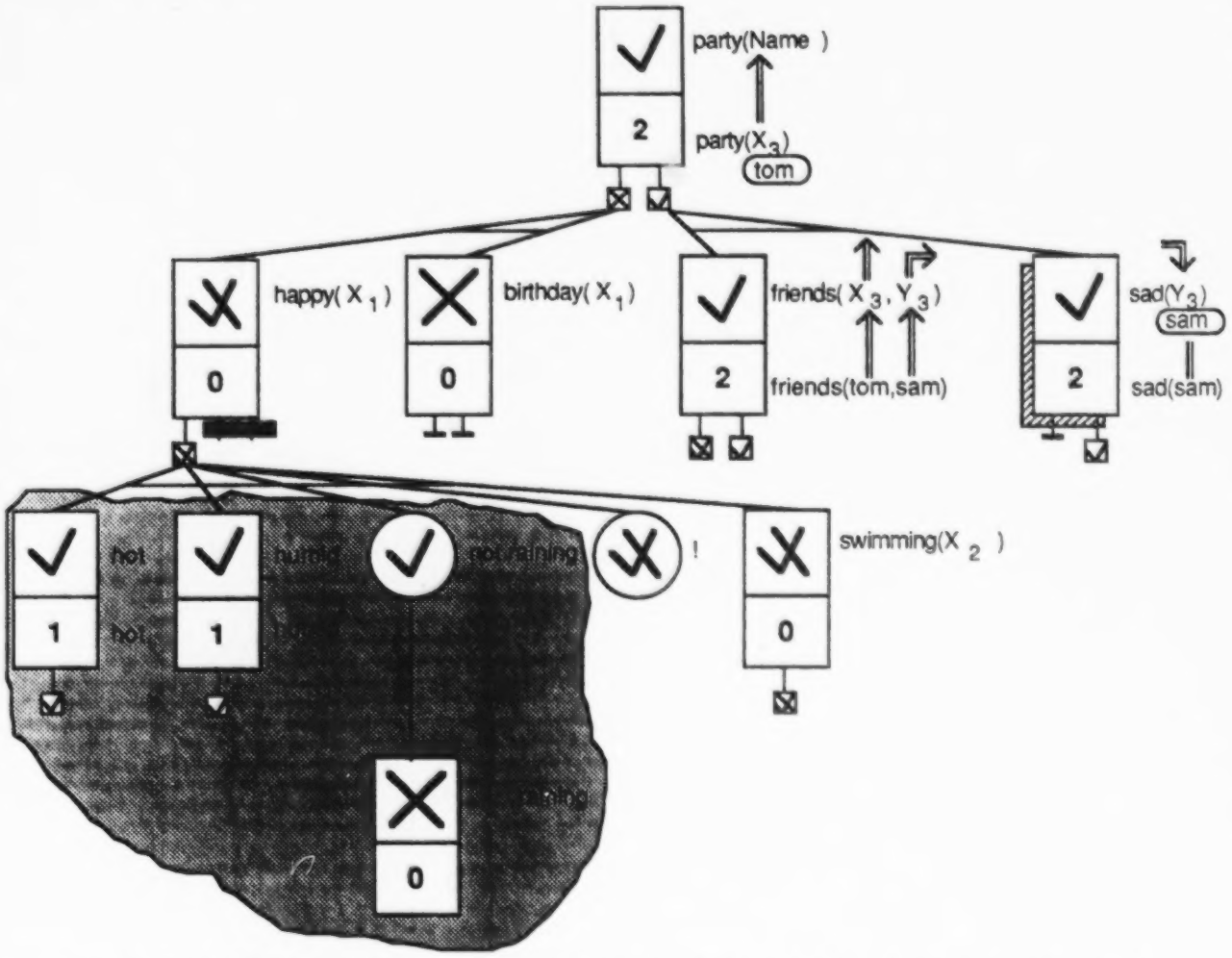

Figure 6 Final AORTA snapshot after the query ?- party(Name). party/1 has just succeeded with Name = tom.

At the instant that **birthday** fails, the 'backtracking point' (the 'youngest surviving heir' of the older sister of the just-failed goal) is the *clause* status box corresponding to clause 1 of **swimming** in figure 6. The technique for backtracking is to identify the backtracking point, then turn that tick into a tick/cross combination, and 'carry on'. Since there are no other clauses for **swimming**, it fails at this point (hence the tick/cross combination in its *procedure* status box), and likewise for the cut. There are no other accessible backtracking points (the 'darkening' of the frozen cloud indicates that an attempt was made to backtrack into it), so clause one of **happy** receives a tick/cross combination to show that it initially won only to fail on backtracking. Clause branches two and three of happy have been eliminated by the cut, (for this invocation only, of course), so the *procedure* status box for happy becomes a tick/cross combination as well. The *clause* status box for clause 1 of **party** therefore receives a cross (failure), and clause two of **party** is now attempted. The **friends** goal succeeds initially on clause one i.e. **friends(tom,john)**, but **sad(john)** fails. After backtracking (which leaves a tick/cross in the *clause* status box for clause 1 of **friends**), now **friends** succeeds on clause two with **friends(tom,sam)**. This leads to a new invocation of the **sad** goal. To indicate that there has been a previous invocation at the same relative point in the search space, a shaded *ghost status box* is drawn. The goal **sad** now succeeds on clause two, as does **party**. Notice in figure 1 that $\mathbf{X_3}$ (and hence **Name**) is instantiated to **tom**, $\mathbf{Y_3}$ is instantiated to **sam** and that this instantiation is passed to the goal **sad**. The goal **sad($\mathbf{Y_3}$)**, with $\mathbf{Y_3}$ instantiated to

sam, matches directly against the fact **sad(sam)** in the database.

The above notational conventions provide a thorough description of Prolog execution. This in turn provides a coherent description for teaching about Prolog as well as a vehicle for implementing practical tracing and debugging facilities for experienced Prolog users. An overview of our Prolog curriculum and a description of our implemented tracing/debugging facilities are presented in the upcoming sections.

3. A textbook, video, and workbook curriculum

The AORTA diagram representation was developed to fulfill a dual need, as stated in section 1. In this section, we want to focus on the way the notation is integrated into our teaching material. Incidentally, in case you have been thinking to yourself, 'Learning Prolog was easy, but this notation is really tricky,' we have at least anecdotal evidence that our students and the users of our graphical tracer feel this way for the first day, and then spontaneously use AORTA diagrams in their own 'scribblings' to help them understand what is really going on.

3.1 Curriculum motivation

We have developed a Prolog course (Eisenstadt et. al., in press) as part of an Alvey/SERC initiative. This initiative identified a need among computing professionals to learn about Prolog in a distance-teaching setting (i.e. in their own environment). We felt that since the 'essentials' of Prolog were adequately covered in Bratko's *Prolog Programming for Artificial Intelligence* (Bratko, 1986), the most appropriate thing to do would be to build upon it by providing detailed diagrammatic 'execution snapshots', re-working some of the more difficult examples, and adding our own emphasis on program design and sophisticated Prolog techniques. We thus concentrated our efforts on clarifying the inner workings of the interpreter, and helping students evolve an execution model which would lead to a clearer (and hopefully less bug-riddled) understanding of Prolog execution. The AORTA notation we had been developing for our advanced graphical debugger appeared to fit the bill, particularly since 'conceptual clarity' was already a high priority objective for that research project.

To encourage wide acceptability of the course, the teaching material works with all the most popular implementations of Prolog, including Quintus Prolog, C-Prolog, LPA MacPROLOG, Arity Prolog, Prolog-2, and even (up to the point where we discuss meta-interpreters and infix notation) Turbo Prolog. For those students who haven't yet decided which commercial Prolog to buy, we include a sample 'core' (Edinburgh syntax) Prolog interpreter on a 5.25" IBM-PC compatible floppy disk so that they can get started immediately. The interpreter is minimal, in that it has no editing environment, but it will run all the examples in the course. We also include full source code listings of all the examples used in the course, both in hard copy and on the course disk.

3.2 Course objectives

Students completing our 80-hour study package are expected to achieve the following objectives:

- Use any of a range of hardware/software configurations to enter, run, and debug sophisticated Prolog programs.
- Explain the practical significance of logic programming terminology (e.g. 'functor', 'unification', 'backtracking').
- Implement a program to determine whether a person is eligible to receive free prescriptions.
- Implement a travel planning program which can propose times, flights, and routes for an airline trip from city A to city B via intermediate cities.
- Improve existing planning programs so that they avoid 'endless loops'
- Implement an expert system shell which can answer 'why' and 'how' queries about its own

reasoning process.

These objectives are addressed through a graded series of presentations and exercises contained in the material outlined in the next sections.

3.3 Course contents

The course comprises a set of course notes, a workbook, a set book, a 120-minute video cassette, and a floppy disk. The main study material is 160 pages in length, and is subdivided as follows (only the top-level headings are shown, along with comments about the underlying pedagogical principles):

Part I: Essential Prolog

1 Prolog in twenty minutes

The basics of Prolog are easy, so students are launched into examples based on airline flight reservations straight away.

2 Program development

Here we introduce the concept of a conservative 'design/edit/invoke Prolog/load/pose queries' cycle

3 Behind the scenes

The declarative and procedural views of Prolog are presented, and a model of Prolog execution, based on AORTA diagrams, is introduced. Exercises include such things as identifying backtracking points by locating nodes in an AORTA diagram, and working through a series of unification problems.

4 Building problem solving programs from scratch

We provide a handle for getting into a good mental style for solving problems in Prolog. A suggested technique involves a series of steps including: identify problem area; consider an elementary concrete case; sketch objects and relations; invent functors and arguments; encode the sketch in terms of functors/arguments; invent problem query; decompose problem into easy and hard variants; add facts or rules to cope with easy variants; add rules to cope with main subproblem; generalise query to take more variables; test program; use richer structures; etc.

5 Understanding recursion

The concept of 'procedure invocation' is explored in detail, and a series of AORTA snapshots is used to explain recursion. Video 1 follows this up in an animated sequence.

Part II: The Power of Prolog Representation

6 Representation style and deductive reasoning

We discuss problems of choosing 'good' functors and arguments, and links to logical deduction.

7 Elementary list manipulation

Basic list processing, **member/2**, and lots of unification exercises are presented.

8 Using structures

We describe more stylistic issues such as data abstraction and 'good representation' involving structures.

9 A closer look at unification

Some subtle cases of unification are described in detail. Bratko's 'Monkey and Banana' problem is illustrated step-by-step to show a particularly tricky case of shared variables becoming instantiated in a counter-intuitive way. Lots of AORTA snapshots are used here.

10 Special list processing techniques

We show how to use an 'accumulator' argument to perform cycle detection, and how to use 'successive instantiation' (building up a list structure in the clause head) to return results. Video 2 amplifies these points with extensive AORTA diagram animations. Students are asked to augment Bratko's travel-planning program so that it deals correctly with cycle detection.

11 Generators and failure-driven loops

A specific set of programming cliches is introduced and explicated.

12 Reshaping the search space: the cut, **not/1**, and **once/1**

We use AORTA diagrams and the family metaphor to show precisely what happens when a cut is encountered, and how this can be used to implement (a form of) negation. Exercises require students to draw 'frozen clouds' and to 'chop off' clause branches on AORTA diagrams to see precisely the effect and scope of the cut. Video 3 uses more AORTA animations to reinforce these concepts.

Part III: An Expert System Shell

13 Choosing a representation and building a basic shell

A 'rational reconstruction' of Bratko's expert system shell is presented, starting from a minimal meta-level interpreter. The accompanying Video 4 shows an expert travel planning program (developed by Expert Systems International for Thomas Cook) in action, and relates it to the shell and travel planning programs used in the course.

14 'How' explanations and truth values

The primitive shell is augmented to deal with 'how' explanations and explicitly-stored negative facts.

15 Other enhancements: query-the-user and 'why'

The shell is spruced up to deal with interactions with the user and a sensible 'why' explanation facility.

The above notes are accompanied by a workbook of exercise material, 48 pages in length. The workbook includes a 'how to get started' guide, which summarizes this differences among the 10 most popular implementations of Prolog. There are lots of 'unfilled' AORTA diagrams for the students to fill in as part of their worked exercises. The accompanying video material (120 minutes) relies extensively on AORTA diagram notation, and presents numerous animated sequences. A separate 16-page set of video notes is used to help the students to work through some of the exemplary problems contained in several stop/start video sequences.

4 The Graphical Tracer Implementation

It is one thing to have a 'paper and pencil' execution model, but quite another to have a consistent implementation of it running on a computer. Normally, there are very different constraints influencing what is appropriate to present in text, on video, and in a working computer-based trace package. Our feeling was that we should present a consistent story line across different media as far as was humanly possible. This section describes the graphical tracer as currently implemented.

4.1 The Long Distance View (LDV)

A practical tracer poses an interesting set of constraints for the Prolog environment designer. The search tree for serious Prolog programs can be enormous, and may thwart the designer who has a nice 'toy' paper and pencil execution model. Our aim has been to make our AORTA diagram the heart of a practical Prolog tracing package aimed at experienced Prolog users. The key lies in the use of a large graphics display and a compressed long-distance view of the AORTA diagram which can be zoomed in upon to provide the kind of detail described above.

The long distance view (LDV) shows the *execution space* of the program (as opposed to the full *search space).* This is done by means of a schematized AND/OR tree in which individual nodes summarize the outcome of a call to a particular procedure. Something must always be sacrificed for the benefit of full global perspective, so we eliminate the clause counter and detailed clause head information (clause branches and clause status boxes) of the AORTA diagrams. This leaves us with a nearly-traditional AND/OR tree, but with enriched 'nodes' in the form of collapsed status boxes. To provide the most meaningful display of information in the smallest space, we rely on colour in our colour-workstation implementation. Thus ticks are green, indicating success, crosses are red, indicating failure, tick/cross combinations are pink ('nearly red'). In the collapsed status box, it is sufficient to display just the colour, e.g. a very small fully-shaded green box instead of a box with a hard-to-see green tick in it. We simulate

this in our black and white display using the correspondence shown in figure 7.

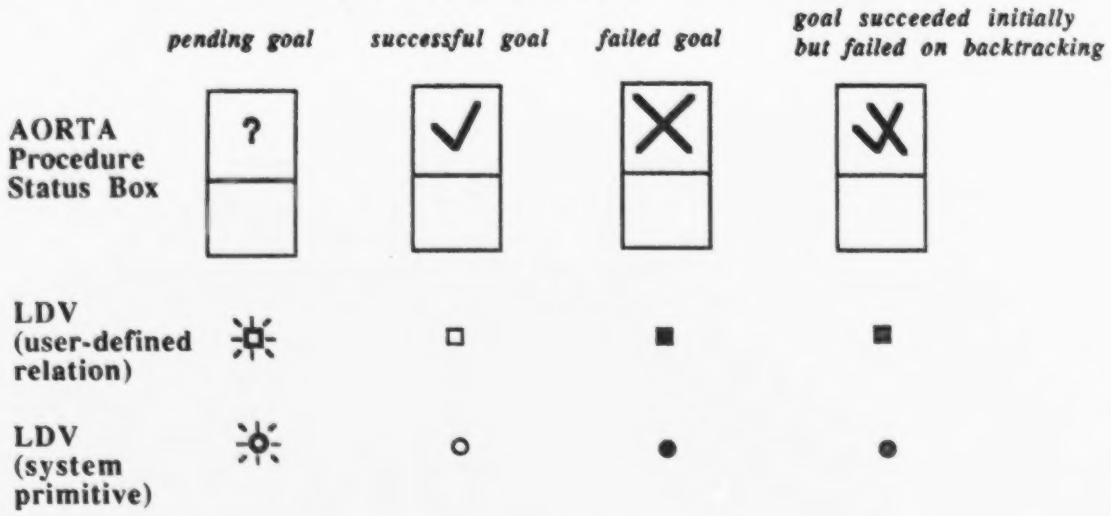

Figure 7 Correspondence between full status boxes and collapsed status boxes, used to simulate colour. Filled in white = green (success); black shading = red (failure); grey shading = pink (success followed by failure on backtracking).

Given the program for **party** in section 2.3, the LDV is shown in figure 8.

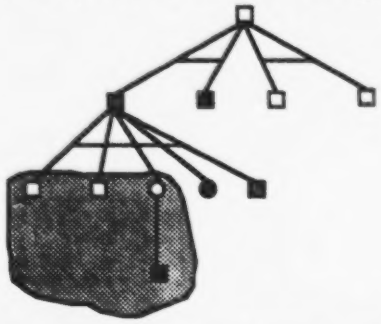

Figure 8 Long Distance View showing the execution of the query ?- **party(Name)**.

The concise manner of the LDV representation facilitates the analysis of very large Prolog programs. At the current resolution of our Apollo workstation displays, we can fit an execution space of 2,500 nodes on a single screen, with the individual status shadings/colours still discriminable. Even if the execution space is too large to be meaningfully displayed within a single graphics pane, the user is able to scroll around the pane, selecting the area of the tree which he or she wishes to look at. A facility will be provided that allows users continually to monitor which part of the tree they are currently looking at.

Associated with such a view are strong notions of gestalt throughout the tree. Similar relations with similar arguments and calling patterns will produce similar shapes of execution space throughout the entire LDV. Likewise certain lumps of code will have characteristic tree 'shapes' associated with them. Put yourself in the position of a programmer who has been developing the associated code over a period of days, and has become accustomed to the repetition of certain familiar shapes. We believe that ascribing functionality to large branches of the tree will be straightforward, and thus, locating 'items of interest' in the tree is surprisingly easy. Such items of interest can, of course, be inspected more closely, even while preserving a considerable degree of surrounding context. In section 4.2 we describe our 'selective highlighting' facility which enables the programmer rapidly to locate nodes in the tree which satisfy some particular constraint or behavioural description.

In the future it is intended that the user may be able to specify various graphical shorthand notations in order to simplify the display. Currently we cater for concepts based on the notion of the *shallow cliche*, e.g. simple tail recursion, as in the classic **member/2**. We hope in the future to be able to provide the user with the facility to abstract arbitrary amounts of code via 'graphical portray' declarations made alongside source code.

4.2 Selective Highlighting

When debugging a program a frequent observation we have made is that it would be nice if we could ask questions like 'where did variable **X** get instantiated to such-and-such' or 'where in the program does **foo** get called by **bar**' and our debugger would immediately tell us. To accommodate this wish we provide a Selective Highlighting option in the LDV display. This option allows you to specify a given relation or argument(s), or combination of relation and arguments for highlighting. This may be further constrained by specifying a similar combination for a parent goal that both child and parent have to satisfy. As a further constraint you can specify a legal Prolog goal that must also be satisfied for the highlight condition to hold. When you specify such a combination, this results in those nodes in the tree that satisfy the conditions being highlighted. For example, suppose we have a bug in a production rule interpreter. The symptoms are that a rule, **rule12**, that we expected to be used is inexplicably overlooked by some goal, say **select_rule**, but only when the calling goal **rule_cycle** has an arbitrary 'debug flag' set. To debug this we need to be able to find out exactly what is going on when this happens. Using the selective highlight facility we can ask to see every time the goal **select_rule** is called with the list of rules **Rule_set** as its argument, by the mother goal **rule_cycle**, with one of its arguments set to be the flag **trace**; however since we're only interested in the times this occurs when the rule **rule12** is in the rule set we can specify the further constraint **member(rule12,Rule_set)**. The result of such an action is to cause the specified items to be highlighted *wherever* this combination occurs in the LDV. The facility allows for rapid location and tracing of given relations or variables. It also allows the user to effectively spy a variable or a particular variable instantiation and observe its behaviour retrospectively in the trace. Using different shapes for highlighting (or colours in our coloured display implementation) several things can be highlighted at once, making it possible, for example, to consider occasions when **rule_cycle** got called without its trace flag set but with **rule12** still in the rule set.

4.3 Replay

The nature of backtracking means that certain aspects of program execution, e.g. new calls to previously failed goals, can best be explain dynamically. We deal with this problem by providing a replay facility whereby the user can see the dynamic execution of the program through the LDV execution space or AORTA diagrams, clearly indicating failure and subsequent backtracking, re-attempting of goals, subsequent failure, resatisfaction or retries. The replay facility thus allows the user to view the execution space at any given time, or at any particular goal invocation. The user can control the speed of the replay with slow motion and single step options being available.

Our replay capability is possible only because we store an exhaustive history of the program's execution. Our experience is that being able to home in quickly on buggy code is sufficiently rewarding to justify the storage overheads of history preservation. Endless loops must, of course, be trapped for example by a depth bound interpreter using stack monitoring (à la Shapiro,1982).

The replay facility allows the user to step through a piece of execution history and reason out what actually happened. This is true especially when the user wants to figure out a complex case of backtracking and wishes to find out where final or interim instantiations came about. The user can tell the program to stop at a particular point, which may either be pre-specified or else indicated via a mouse click 'live' during replay. The utility thus allows the user to effectively freeze the program at any step of its execution and

use any of the other existing tools on the program at that point, or to step backwards or forwards one more 'frame', or to carry on with the execution.

The user can also watch the program re-run as many times as necessary, but because we display the execution space continually the user can watch backtracking and its scope 'live' as it happens. In the event of extensive backtracking which eventually succeeds we can imagine seeing large areas of successful nodes (green or white) turn to failure (red or black) until finally the process stops and the successful nodes starts to regain their lost territory. The whole process can be stopped and frozen at an instant to allow for closer inspection by a mouse click. What is also important here is for the *gestalt* to be right, i.e. for the display not only to be informative, but also to feel like an accurate version of the underlying Prolog machine.

In the preceding section we introduced the notion of a *ghost status box*, where the ghost indicated that a previous or future invocation of a goal in the same place in the execution space has/will take place. It is possible to turn the time clock back to that point where the ghost goal was about to be attempted by simply clicking on the shaded part of its status box. In this way information about backtracking through conjuncts (e.g. in a permutation algorithm where you have a generator and test relations as sister goals) may be readily gained.

4.4 Zooming

Zooming links the LDV and AORTA representations by allowing the user to switch between the two, enabling the user to zoom in on the execution/unification history of the clause from the LDV. In this way the complementary nature of the two views is emphasised. The LDV allows the user to view the program from a clear perspective, uncluttered by unwanted information, and explore its behaviour in an orderly, informed manner. Switching from the LDV of a particular node to the AORTA view thus increases the amount and type of information available to the user. Not only is this AORTA view useful for teaching novices, but it also (in conjunction with the LDV and zooming) allows the expert rapid and easy access to a large amount of unification history and surrounding information.

The 'close-up' view shows not only the 'focussed upon goal', but also that goal's mother, step-fathers, sisters, and children. It thus provides both clarity of detail and some surrounding context within which to perform debugging. We call this our 'three ply close up view', because three generations of goals are visible at once. The replay facilities described in section 4.3 are also available in the close up view, which means that the precise details of the unification history can be observed if necessary.

Since zooming and highlighting requests always begin with the LDV, all the perspective information associated with the LDV is available at the point of choice, allowing the user clearly to understand the context of the code which is being observed 'close up', thus removing the 'forest-vs-trees' problem associated with conventional 'spy' packages. In such packages, once a 'spied' goal is reached it may no longer be clear how you arrived there, how the instantiations of the variables have been derived, what state the program is in, what side-effects have taken place, whether the program has only reached this point on backtracking, and (if a 'redo' is involved) the nature, cause, and scope of the backtracking involved. By the combination of the LDV, AORTA, zooming, selective highlighting and replay facilities, the user may more readily understand the state of the program, and thus arrive rapidly at the source of problematic bugs.

4.5 A worked example of the graphical tracer

The following program is a buggy quicksort:

```
qsort([],[]).                          %the sort of an empty list is an empty list
qsort([X|Xs],Result):-                 %sort list with head X, tail Xs, to give Result
        split(Xs,X,Lo,Hi),             %split Xs around X intoLower and Higher
        qsort(Lo,Sorted_Lo),           %recursively sort Lo list into Sorted_Lo
        qsort(Hi,Sorted_Hi),           %recursively sort Hi list into Sorted_Hi
        append(Sorted_Lo,[X|Sorted_Hi],Result). %glue together in Result

split([X|Xs],Crit,[X|Lo],Hi):-         %X is head of Lo list if
        X < Crit,                      %X is lower than the split criterion
        split(Xs,Crit,Lo,Hi).          %recursively split rest of list
split([X|Xs],Crit,Lo,[X|Hi]):-         %X is head of Hi list if
        X > Crit,                      %X is greater than split criterion
        split(Xs,Crit,Lo,Hi).          %recursively split rest of list

append([X|Xs],Y,[X|Zs]):-              %place head on front of output list
        append(Xs,Y,Zs).               %process rest of Xs recursively
append([],Ys,Ys).                      %stopping condition,leave Ys alone
```

If we pose the query **?- qsort([2,1,3],P).** to the Prolog top-level, then the query will fail. Invoking the graphical tracer, the user is presented with the LDV shown in figure 9.

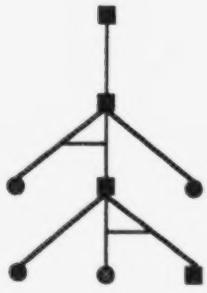

Figure 9 LDV for the query **qsort([2,1,3],P).**

From the above diagram we can immediately see that the top goal **qsort** only attempted one sub-goal. This must have been its first, **split**. Now as **split** only calls itself or the primitives **>** and **<**, they are clearly the rest of the tree that we're seeing, thus it must be one of them that's causing the error. To confirm this we can highlight all occurrences of the goal **split** called by the goal **split** itself, i.e. all the recursive invocations of **split**. This is illustrated in figure 10.

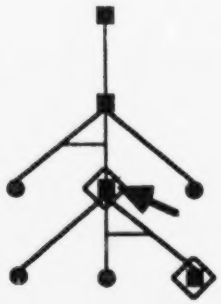

Figure 10. Selective highlighting of all occurrences of the goal **split** called by the goal **split** (i.e. just the recursive calls). The arrow (which is not part of the LDV) shows the node on which we're about to zoom.

From figure 10 there are a large number of things that we can observe about the program. We can see that there were two recursive calls of **split**. Notice also that we can see a characteristic shape for the **split** goal: it calls a primitive in one or the other of its clauses, and if that is successful calls itself, leaving a characteristic shape that we could spot in a much larger trace if we knew **split** was to be called. The initially successful primitive goals are plain by their shading in the diagram, i.e. they must have failed on backtracking. In such circumstances we can always also identify the source of such backtracking: it must be the youngest sister of the goals that are grey-shaded, or the youngest subgoal of that goal following on down the tree. Thus from the LDV alone we can spot that it must be the youngest **split** goal that failed and caused the backtracking that led to the failure of the **qsort** goal. To confirm this and to shed more light on the actual cause of the failure we can zoom in on the node. Since the suspect doesn't have any children, we might as well zoom on the parent node to obtain more contextual information. This is the node to which the cursor arrow is pointing in figure 10. Our selective highlighting facility allows us to 'freeze' the action at the moment just *before* the first failure of **split**. The result of zooming is shown in figure 11. The 'faded out' lines depict the execution space which will be traversed later.

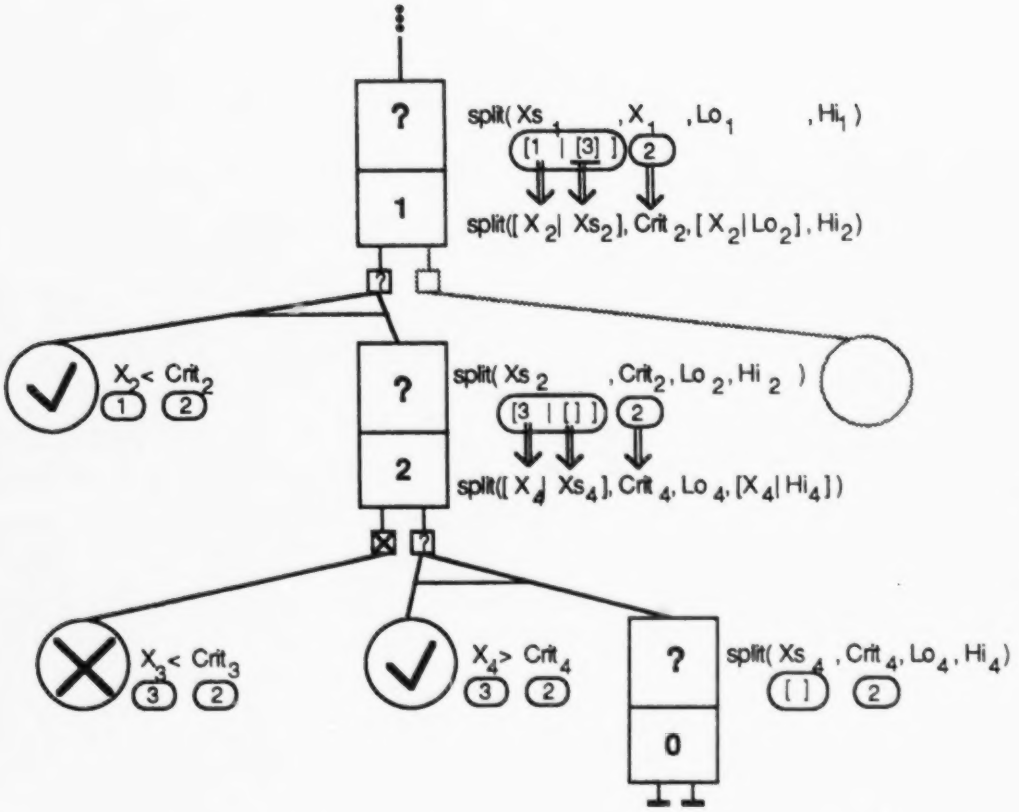
?
split(Xs 1 , X 1 , Lo 1 , Hi 1)
[1 | [3]]
2
1
split([X 2| Xs 2], Crit 2, [X 2| Lo 2] , Hi 2)
X 2< Crit 2
1
2
?
split(Xs 2 , Crit 2, Lo 2, Hi 2)
[3 | [ ]]
2
2
split([X 4| Xs 4], Crit 4, Lo 4, [X 4| Hi 4])
X 3< Crit 3
3
2
X 4> Crit 4
3
2
?
0
split(Xs 4 , Crit 4, Lo 4, Hi 4)
[ ]
2

Figure 11 AORTA produced by a zoom on the node pointed to by the arrow in figure 10. This node is the **split** node in the middle of the diagram.

The AORTA diagram reveals that the youngest **split** node, the one which we suspected from the LDV, failed to unify on either of its two clauses. Thus we can see that the null case of **split** is not catered for, leading to the failure of the goal and the subsequent failure of the primary goal **qsort([2,1,3],P)**. The AORTA zoom thus confirms our suspicions from the LDV that the final **split** goal was the cause of the failure, and further rapidly indicated the reason for this failure.

In our current implementation, only relation names are shown next to the procedure status boxes, with argument and unification details available as 'popup' options. More detailed examples, including traces and replays involving an expert system and a small compiler (involving much larger execution spaces), are described in Eisenstadt & Brayshaw (1986).

5 Conclusions

Our aim has been to reconcile a global view of Prolog program execution with the 'truth' about unification and clause selection. Moreover, we have wanted to satisfy both the needs of novices learning Prolog and the needs of expert Prolog users writing and debugging very large programs. These intentions have, in turn, led us along a pathway of 'creative tension' to satisfy somewhat contradictory constraints. The key ingredients of our approach have been (i) appreciation of the power of gestalt patterns, (ii) recognition of the need (and the ability) to display thousands of nodes at a time, (iii) enhancement of traditional AND/OR tree branches with individual clause details, (iv) enhancement of and/or tree nodes with goal 'status boxes' and (v) ability to vary the type of detail being investigated with the particular grain size, rather than using a

physical 'zoom'.

The result of our effort has been a complete 'cradle-to-grave' environment in the form of text, video, and a computer-implemented trace package. The integrated environment conveys a consistent 'party line' in a manner that makes execution intuitively clear and at the same time provides real hands-on debugging power for experienced users. We eagerly await empirical studies of this integrated environment to analyse its real effectiveness.

6 References.

Bratko I., *Prolog Programming for Artificial Intelligence*. Reading, MA: Addison-Wesley, 1986.

Brown M.H., and Sedgewick R. Progress Report: Brown University Instructional Computing Laboratory, *ACM SIGCSE Bulletin 16, 1,(February 1984)*

Bundy, A., Pain, H., Brna, P., and Lynch, L. A proposed Prolog story. DAI Research Paper 283, Department of Artificial Intelligence, University of Edinburgh, 1986.

Byrd, L. Understanding the control flow of Prolog programs. In S-Å Tarnlund (Ed.), *Proceedings of the 1980 Logic Programming Workshop*, pp. 127-138, 1980.

Carroll J.M. & Thomas J.C. Metaphor and the Cognitive Representation of Computing Systems, *IEEE Transactions on systems, man, and cybernetics*, SMC-12(2), 1982.

Clocksin W.F., and Mellish C.S. *Programming in Prolog (First Edition)*, New York: Springer-Verlag 1981.

Clocksin W.F., and Mellish C.S. *Programming in Prolog (Second Edition)*, New York: Springer-Verlag 1984.

Coombs M.J., and Stell, J.G. A model for debugging Prolog by symbolic execution: the separation of specification and procedure. Research Report MMIGR137, Department of Computer Science, University of Strathclyde, 1985.

DuBoulay, J.B.H., O'Shea, T., and Monk, J. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man Machine Studies*, 14 (3), 237-249, 1981.

Eisenstadt, M., and Brayshaw, M. The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, in press. Also available as Technical Report no. 21, Human Cognition Research Laboratory, The Open University, Milton Keynes, 1986.

Eisenstadt, M., Dixon, M., Kahney, H., and Kriwaczek, F. *Intensive Prolog*. Milton Keynes, U.K.: Open University Press (in press).

Halasz F., and Morán T.P.. Analogy Considered Harmful, in *Proceedings of Human factors in Computing Systems*, Gaithersburg, Maryland, March 15-17th., 1982.

Hardy S. A new software environment for list-processing and logic programming, in T. O'Shea, and M. Eisenstadt (Eds.), *Artificial Intelligence: Tools, Techniques, and Applications*. New York: Harper & Row, 1984.

Rajan T.. APT: A Principled Design for an Animated View of Program Execution for Novice Programmers, Technical Report No. 19, Human Cognition Research Laboratory, Open

University, Milton Keynes, December, 1986.

Shapiro E.Y. *Algorithmic Program Debugging*. Cambridge, MA: MIT Press, 1982.

Stirling L. and Shapiro E.Y. *The Art of Prolog: Advanced Programming Techniques*, Cambridge, MA: MIT Press, 1986.

Taylor J.,Programming in Prolog: An in-depth study of problems for beginners learning to program in Prolog. PhD. Thesis, Cognitive Studies Programme, University of Sussex, 1987.